1. SECTION1
    1. Sorting
    2. Design Patterns for Sorting
    3. Sorting an Array
2. Graphical Convolution Algorithm
3. Algorithm Overview

6. Sorting

6.1. Basic sort algorithms

(From Wikipedia, the free encyclopedia)

In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation, or reordering, of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being invented to this day (for example, library sort was first published in 2004). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide-and-conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

Classification

Sorting algorithms used in computer science are often classified by:

- Computational complexity (worst, average and best behaviour) of element comparisons in terms of the size of the list (n). For typical sorting algorithms good behavior is O(n log n) and bad behavior is $\Omega(n^2)$. (See Big O notation) Ideal behavior for a sort is O(n). Sort algorithms which only use an abstract key comparison operation always need at least Ω(n log n) comparisons on average.
- Computational complexity of swaps (for "in place" algorithms).
- Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in place", such that only O(1) or O(log n) memory is needed beyond the items being sorted, while others need to create auxiliary locations for data to be temporarily stored.
- Recursion. Some algorithms are either recursive or non recursive, while others may be both (e.g., merge sort).
- Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). See below for more information.
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- General method: insertion, exchange, selection, merging, etc. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.

Stability

Stable sorting algorithms maintain the relative order of records with equal keyshttp://en.wikipedia.org/wiki/Strict_weak_ordering (i.e. sort key values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first coordinate:

(4, 1) (3, 7) (3, 1) (5, 6)

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

(3, 7) (3, 1) (4, 1) (5, 6) (order maintained)

(3, 1) (3, 7) (4, 1) (5, 6) (order changed)

Unstable sorting algorithms may change the relative order of records with equal keys, but stable sorting algorithms never do so. Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original data order as a tie-breaker. Remembering this order, however, often involves an additional space cost.

Sorting based on a primary, secondary, tertiary, etc. sort key can be done by any sorting method, taking all sort keys into account in comparisons (in other words, using a single composite sort key). If a sorting method is stable, it is also possible to sort multiple times, each time with one sort key. In that case the sort keys can be applied in any order, where some key orders may lead to a smaller running time.

6.1.1. Insertion sort

(From Wikipedia, the free encyclopedia)

Insertion sort is a simple sorting algorithm, a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort, but it has various advantages:

- Simple to implement
- Efficient on (quite) small data sets
- Efficient on data sets which are already substantially sorted: it runs in O(n + d) time, where d is the number of inversions
- More efficient in practice than most other simple O(n2) algorithms such as selection sort or bubble sort: the average time is n2/4 and it is

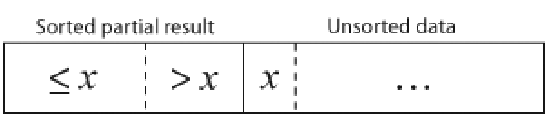
linear in the best case

- Stable (does not change the relative order of elements with equal keys)
- In-place (only requires a constant amount O(1) of extra memory space)
- It is an online algorithm, in that it can sort a list as it receives it.

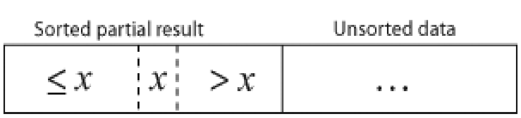
Algorithm

In abstract terms, every iteration of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input. The choice of which element to remove from the input is arbitrary and can be made using almost any choice algorithm.

Sorting is typically done in-place. The resulting array after k iterations contains the first k entries of the input array and is sorted. In each step, the first remaining entry of the input is removed, inserted into the result at the right position, thus extending the result:



becomes:



with each element > x copied to the right as it is compared against x.

The most common variant, which operates on arrays, can be described as:

1. Suppose we have a method called insert designed to insert a value into a sorted sequence at the beginning of an array. It operates by starting at

the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. It has the side effect of overwriting the value stored immediately after the sorted sequence in the array.

2. To perform insertion sort, start at the left end of the array and invoke insert to insert each element encountered into its correct position. The ordered sequence into which we insert it is stored at the beginning of the array in the set of indexes already examined. Each insertion overwrites a single value, but this is okay because it's the value we're inserting.

A simple pseudocode version of the complete algorithm follows, where the arrays are zero-based:

```
insertionSort(array A)

for i <- 1 to length[A]-1 do

value <- A[i]

j <- i-1

while j >= 0 and A[j] > value do

A[j + 1] = A[j];

j <- j-1

A[j+1] <- value
```

Good and bad input cases

In the best case of an already sorted array, this implementation of insertion sort takes O(n) time: in each iteration, the first remaining element of the input is only compared with the last element of the sorted subsection of the array. This same case provides worst-case behavior for non-randomized and poorly implemented quicksort, which will take O(n2) time to sort an

already-sorted list. Thus, if an array is sorted or nearly sorted, insertion sort will significantly outperform quicksort.

The worst case is an array sorted in reverse order, as every execution of the inner loop will have to scan and shift the entire sorted section of the array before inserting the next element. Insertion sort takes O(n2) time in this worst case as well as in the average case, which makes it impractical for sorting large numbers of elements. However, insertion sort's inner loop is very fast, which often makes it one of the fastest algorithms for sorting small numbers of elements, typically less than 10 or so.

Comparisons to other sorts

Insertion sort is very similar to selection sort. Just like in selection sort, after k passes through the array, the first k elements are in sorted order. For selection sort, these are the k smallest elements, while in insertion sort they are whatever the first k elements were in the unsorted array. Insertion sort's advantage is that it only scans as many elements as it needs to in order to place the k + 1st element, while selection sort must scan all remaining elements to find the absolute smallest element.

Simple calculation shows that insertion sort will therefore usually perform about half as many comparisons as selection sort. Assuming the k + 1st element's rank is random, it will on the average require shifting half of the previous k elements over, while selection sort always requires scanning all unplaced elements. If the array is not in a random order, however, insertion sort can perform just as many comparisons as selection sort (for a reverse-sorted list). It will also perform far fewer comparisons, as few as n - 1, if the data is pre-sorted, thus insertion sort is much more efficient if the array is already sorted or "close to sorted." It can be seen as an advantage for some real-time applications that selection sort will perform identically regardless of the order of the array, while insertion sort's running time can vary considerably.

While insertion sort typically makes fewer comparisons than selection sort, it requires more writes because the inner loop can require shifting large

sections of the sorted portion of the array. In general, insertion sort will write to the array O(n2) times while selection sort will write only O(n) times. For this reason, selection sort may be better in cases where writes to memory are significantly more expensive than reads, such as EEPROM or Flash memory.

Some divide-and-conquer algorithms such as quicksort and mergesort sort by recursively dividing the list into smaller sublists which are then sorted. A useful optimization in practice for these algorithms is to switch to insertion sort for "sorted enough" sublists on which insertion sort outperforms the more complex algorithms. The size of list for which insertion sort has the advantage varies by environment and implementation, but is typically around 8 to 20 elements.

Variants

D.L. Shell made substantial improvements to the algorithm, and the modified version is called Shell sort. It compares elements separated by a distance that decreases on each pass. Shell sort has distinctly improved running times in practical work, with two simple variants requiring O(n3/2) and O(n4/3) time.

If comparisons are very costly compared to swaps, as is the case for example with string keys stored by reference or with human interaction (such as choosing one of a pair displayed side-by-side), then using binary insertion sort can be a good strategy. Binary insertion sort employs binary search to find the right place to insert new elements, and therefore performs

$$\lceil log_2(n!) \rceil$$

comparisons in the worst case, which is Θ(n log n). The algorithm as a whole still takes Θ(n2) time on average due to the series of swaps required for each insertion, and since it always uses binary search, the best case is no longer Ω(n) but Ω(n log n).

To avoid having to make a series of swaps for each insertion, we could instead store the input in a linked list, which allows us to insert and delete elements in constant time. Unfortunately, binary search on a linked list is impossible, so we still spend O(n2) time searching. If we instead replace it by a more sophisticated data structure such as a heap or binary tree, we can significantly decrease both search and insert time. This is the essence of heap sort and binary tree sort.

In 2004, Bender, Farach-Colton, and Mosteiro published a new variant of insertion sort called library sort or gapped insertion sort that leaves a small number of unused spaces ("gaps") spread throughout the array. The benefit is that insertions need only shift elements over until a gap is reached. Surprising in its simplicity, they show that this sorting algorithm runs with high probability in O(n log n) time.

Examples

c++ Example:

```
#include <iostream>

#include <cstdio>

//Originally Compiled tested with g++ on Linux

using namespace std;

bool swap(int&, int&); //Swaps Two Ints

void desc(int* ar, int); //Nothing Just Shows The Array Visually

int ins_sort(int*, int); //The Insertion Sort Function

int main()

{
```

```
int array[9] = {4, 3, 5, 1, 2, 0, 7, 9, 6}; //The Original Array

desc(array, 9);

*array = ins_sort(array, 9);

cout << "Array Sorted Press Enter To Continue and See the Resultant
Array" << endl

<< "-----------------8<------------------------------------->8--------------";

getchar();

desc(array, 9);

getchar();

return 0;

}

int ins_sort(int* array, int len)

{

for (int i = 0; i < len; i++)

{

int val = array[i];

int key = i;

cout << "key(Key) = " << key << "\tval(Value) = " << val << endl;

for (; key >= 1 && array[key-1] >= val; --key)

{
```

```
cout << "Swapping Backward\tfrom (key) " << key << " of (Value) " <<
array[key] << "\tto (key) " << key-1

<< " of (Value) " << array[key-1];

cout << "\n\t" << key << " <----> " << key-1 << "\t( " << array[key] << "<--
--> " << array[key-1] << " )";

swap(array[key], array[key-1]);

desc(array, 9);

}

}

return *array;

}

bool swap(int& pos1, int& pos2)

{

int tmp = pos1;

pos1 = pos2;

pos2 = tmp;

return true;

}

void desc(int* ar, int len)

{

cout << endl << "Describing The Given Array" << endl;
```

```
for (int i = 0; i < len; i++)

cout << "________" << "\t";

cout << endl;

for (int i = 0; i < len; i++)

cout << " | " << i << " | " << "\t";

cout << endl;

for (int i = 0; i < len; i++)

cout << " ( " << ar[i] << " ) " << "\t";

cout<<endl;

for (int i = 0; i < len; i++)

cout << "-------" << "\t";

getchar();

}
```

Python Example:

```
def insertion_sort(A):

for i in range(1, len(A)):

key = A[i]

j = i-1

while(j >= 0 and A[j] > key):

A[j+1] = A[j]
```

j = j-1

A[j+1] = key

6.1.2. Selection sort

(From Wikipedia, the free encyclopedia)

Selection sort is a sorting algorithm, specifically an in-placecomparison sort. It has Θ(n2) complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations. It works as follows:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for remainder of the list (starting at the second position)

Effectively, we divide the list into two parts: the sublist of items already sorted, which we build up from left to right and is found at the beginning, and the sublist of items remaining to be sorted, occupying the remainder of the array.

Here is an example of this sort algorithm sorting five elements:

31 25 12 22 11

11 25 12 22 31

11 12 25 22 31

11 12 22 25 31

Selection sort can also be used on list structures that make add and remove efficient, such as a linked list. In this case it's more common to remove the

{

minimum element from the remainder of the list, and then insert it at the end of the values sorted so far. For example:

31 25 12 22 11

11 31 25 12 22

11 12 31 25 22

11 12 22 31 25

11 12 22 25 31

Implementation

The following is a C/C++ implementation, which makes use of a swap function:

```
void selectionSort(int a[], int size)

{

int i, j, min;

for (i = 0; i < size - 1; i++)

{

min = i;

for (j = i+1; j < size; j++)

{

if (a[j] < a[min])

{
```

```
min = j;

}

}

swap(a[i], a[min]);

}

}
```

Python example:

```
def selection_sort(A):

for i in range(0, len(A)-1):

min = A[i]

pos = i

for j in range(i+1, len(A)):

if( A[j] < min ):

min = A[j]

pos = j

A[pos] = A[i]

A[i] = min
```

Analysis

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting

the lowest element requires scanning all n elements (this takes n - 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n - 1 elements and so on, for (n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2 = Θ(n2) comparisons (see arithmetic progression). Each of these scans requires one swap for n - 1 elements (the final element is already in place). Thus, the comparisons dominate the running time, which is Θ(n2).

Comparison to other Sorting Algorithms

Among simple average-case Θ(n2) algorithms, selection sort always outperforms bubble sort and gnome sort, but is generally outperformed by insertion sort. Insertion sort is very similar in that after the kth iteration, the first k elements in the array are in sorted order. Insertion sort's advantage is that it only scans as many elements as it needs to in order to place the k + 1st element, while selection sort must scan all remaining elements to find the k + 1st element.

Simple calculation shows that insertion sort will therefore usually perform about half as many comparisons as selection sort, although it can perform just as many or far fewer depending on the order the array was in prior to sorting. It can be seen as an advantage for some real-time applications that selection sort will perform identically regardless of the order of the array, while insertion sort's running time can vary considerably. However, this is more often an advantage for insertion sort in that it runs much more efficiently if the array is already sorted or "close to sorted."

Another key difference is that selection sort always performs Θ(n) swaps, while insertion sort performs Θ(n2) swaps in the average and worst cases. Because swaps require writing to the array, selection sort is preferable if writing to memory is significantly more expensive than reading, such as when dealing with an array stored in EEPROM or Flash.

Finally, selection sort is greatly outperformed on larger arrays by Θ(nlog n) divide-and-conquer algorithms such as quicksort and mergesort. However, insertion sort or selection sort are both typically faster for small arrays (ie

less than 10-20 elements). A useful optimization in practice for the recursive algorithms is to switch to insertion sort or selection sort for "small enough" sublists.

Variants

Heapsort greatly improves the basic algorithm by using an implicitheapdata structure to speed up finding and removing the lowest datum. If implemented correctly, the heap will allow finding the next lowest element in Θ(log n) time instead of Θ(n) for the inner loop in normal selection sort, reducing the total running time to Θ(n log n).

A bidirectional variant of selection sort, called cocktail sort, is an algorithm which finds both the minimum and maximum values in the list in every pass. This reduces the number of scans of the list by a factor of 2, eliminating some loop overhead but not actually decreasing the number of comparisons or swaps. Note, however, that cocktail sort more often refers to a bidirectional variant of bubble sort.

Selection sort can be implemented as a stable sort. If, rather than swapping in step 2, the minimum value is inserted into the first position (that is, all intervening items moved down), the algorithm is stable. However, this modification leads to Θ(n2) writes, eliminating the main advantage of selection sort over insertion sort, which is always stable.

6.1.3. Bubble sort

(From Wikipedia, the free encyclopedia)

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (i.e. the beginning) of the list via the swaps. (Another opinion: it gets its name from

the way greater elements "bubble" to the end.) Because it only uses comparisons to operate on elements, it is a comparison sort. This is the easiest comparison sort to implement.

A simple way to express bubble sort in pseudocode is as follows:

procedure bubbleSort(A : list of sortable items) defined as:

do

swapped := false

for each i in 0 to length(A) - 2 do:

if A[i] > A[i + 1] then

swap(A[i], A[i + 1])

swapped := true

end if

end for

while swapped

end procedure

The algorithm can also be expressed as:

procedure bubbleSort(A : list of sortable items) defined as:

for each i in 1 to length(A) do:

for each j in length(A) downto i + 1 do:

if A[j] < A[j - 1] then

swap(A[j], A[j - 1])

end if

end for

end for

end procedure

This difference between this and the first pseudocode implementation is discussed later in the article.

Analysis

Best-case performance

Bubble sort has best-case complexity Ω(n). When a list is already sorted, bubblesort will pass through the list once, and find that it does not need to swap any elements. Thus bubble sort will make only n comparisons and determine that list is completely sorted. It will also use considerably less time than $O(n^2)$ if the elements in the unsorted list are not too far from their sorted places. MKH...

Rabbits and turtles

The positions of the elements in bubble sort will play a large part in determining its performance. Large elements at the top of the list do not pose a problem, as they are quickly swapped downwards. Small elements at the bottom, however, as mentioned earlier, move to the top extremely slowly. This has led to these types of elements being named rabbits and turtles, respectively.

Various efforts have been made to eliminate turtles to improve upon the speed of bubble sort. Cocktail sort does pretty well, but it still retains O(n2) worst-case complexity. Comb sort compares elements large gaps apart and

can move turtles extremely quickly, before proceeding to smaller and smaller gaps to smooth out the list. Its average speed is comparable to faster algorithms like Quicksort.

Alternative implementations

One way to optimize bubble sort is to note that, after each pass, the largest element will always move down to the bottom. During each comparison, it is clear that the largest element will move downwards. Given a list of size n, the nth element will be guaranteed to be in its proper place. Thus it suffices to sort the remaining n - 1 elements. Again, after this pass, the n - 1th element will be in its final place.

In pseudocode, this will cause the following change:

procedure bubbleSort(A : list of sortable items) defined as:

n := length(A)

do

swapped := false

n := n - 1

for each i in 0 to n do:

if A[i] > A[i + 1] then

swap(A[i], A[i + 1])

swapped := true

end if

end for

while swapped

end procedure

We can then do bubbling passes over increasingly smaller parts of the list. More precisely, instead of doing n2 comparisons (and swaps), we can use only n + (n-1) + (n-2) + ... + 1 comparisons. This sums up to n(n + 1) / 2, which is still O(n2), but which can be considerably faster in practice.

In practice

Although bubble sort is one of the simplest sorting algorithms to understand and implement, its O(n2) complexity means it is far too inefficient for use on lists having more than a few elements. Even among simple O(n2) sorting algorithms, algorithms like insertion sort are usually considerably more efficient.

Due to its simplicity, bubble sort is often used to introduce the concept of an algorithm, or a sorting algorithm, to introductory computer science students. However, some researchers such as Owen Astrachan have gone to great lengths to disparage bubble sort and its continued popularity in computer science education, recommending that it no longer even be taught.

The Jargon file, which famously calls bogosort "the archetypical perversely awful algorithm", also calls bubble sort "the generic bad algorithm". Donald Knuth, in his famous The Art of Computer Programming, concluded that "the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems", some of which he discusses therein.

Bubble sort is asymptotically equivalent in running time to insertion sort in the worst case, but the two algorithms differ greatly in the number of swaps necessary. Experimental results such as those of Astrachan have also shown that insertion sort performs considerably better even on random lists. For these reasons many modern algorithm textbooks avoid using the bubble sort algorithm in favor of insertion sort.

Bubble sort also interacts poorly with modern CPU hardware. It requires at least twice as many writes as insertion sort, twice as many cache misses, and asymptotically more branch mispredictions. Experiments by Astrachan sorting strings in Java show bubble sort to be roughly 5 times slower than insertion sort and 40% slower than selection sort.

6.2. Effectively sorting algorithms

6.2.1. Shell sort

(From Wikipedia, the free encyclopedia)

Shell sort is a sorting algorithm that is a generalization of insertion sort, with two observations:

- insertion sort is efficient if the input is "almost sorted", and
- insertion sort is typically inefficient because it moves values just one position at a time.

Implementation

The original implementation performs Θ(n2) comparisons and exchanges in the worst case. A minor change given in V. Pratt's book improved the bound to O(n log2 n). This is worse than the optimal comparison sorts, which are O(n log n).

Shell sort improves insertion sort by comparing elements separated by a gap of several positions. This lets an element take "bigger steps" toward its expected position. Multiple passes over the data are taken with smaller and smaller gap sizes. The last step of Shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted.

Consider a small value that is initially stored in the wrong end of the array. Using an O(n2) sort such as bubble sort or insertion sort, it will take roughly n comparisons and exchanges to move this value all the way to the other end of the array. Shell sort first moves values using giant step sizes, so a small value will move a long way towards its final position, with just a few comparisons and exchanges.

One can visualize Shellsort in the following way: arrange the list into a table and sort the columns (using an insertion sort). Repeat this process, each time with smaller number of longer columns. At the end, the table has only one column. While transforming the list into a table makes it easier to visualize, the algorithm itself does its sorting in-place (by incrementing the index by the step size, i.e. using i += step_size instead of i++).

For example, consider a list of numbers like [13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10]. If we started with a step-size of 5, we could visualize this as breaking the list of numbers into a table with 5 columns. This would look like this:

13 14 94 33 82

25 59 94 65 23

45 27 73 25 39

10

We then sort each column, which gives us

10 14 73 25 23

13 27 94 33 39

25 59 94 65 82

45

When read back as a single list of numbers, we get [10 14 73 25 23 13 27 94 33 39 25 59 94 65 82 45]. Here, the 10 which was all the way at the end,

has moved all the way to the beginning. This list is then again sorted using a 3-gap sort, and then 1-gap sort (simple insertion sort).

Gap sequence

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

The shellsort algorithm in action

The gap sequence is an integral part of the shellsort algorithm. Any increment sequence will work, so long as the last element is 1. The algorithm begins by performing a gap insertion sort, with the gap being the first number in the gap sequence. It continues to perform a gap insertion sort for each number in the sequence, until it finishes with a gap of 1. When the gap is 1, the gap insertion sort is simply an ordinary insertion sort, guaranteeing that the final list is sorted.

The gap sequence that was originally suggested by Donald Shell was to begin with N / 2 and to halve the number until it reaches 1. While this sequence provides significant performance enhancements over the quadratic algorithms such as insertion sort, it can be changed slightly to further decrease the average and worst-case running times. Weiss' textbook[4] demonstrates that this sequence allows a worst case O(n2) sort, if the data is initially in the array as (small_1, large_1, small_2, large_2, ...) - that is, the upper half of the numbers are placed, in sorted order, in the even index locations and the lower end of the numbers are placed similarly in the odd indexed locations.

Perhaps the most crucial property of Shellsort is that the elements remain k-sorted even as the gap diminishes. For instance, if a list was 5-sorted and then 3-sorted, the list is now not only 3-sorted, but both 5- and 3-sorted. If this were not true, the algorithm would undo work that it had done in previous iterations, and would not achieve such a low running time.

Depending on the choice of gap sequence, Shellsort has a proven worst-case running time of O(n2) (using Shell's increments that start with 1/2 the array size and divide by 2 each time), O(n3 / 2) (using Hibbard's increments of 2k − 1), O(n4 / 3) (using Sedgewick's increments of 9(4i) − 9(2i) + 1, or 4i + 1 + 3(2i) + 1), or O(nlog2n), and possibly unproven better running times. The existence of an O(nlogn) worst-case implementation of Shellsort remains an open research question.

The best known sequence is 1, 4, 10, 23, 57, 132, 301, 701. Such a Shell sort is faster than an insertion sort and a heap sort, but if it is faster than a quicksort for small arrays (less than 50 elements), it is slower for bigger arrays. Next gaps can be computed for instance with :

nextgap = round(gap * 2.3)

Shell sort algorithm in C/C++

Shell sort is commonly used in programming languages; this is an implementation of the algorithm in C/C++ for sorting an array of integers. The increment sequence used in this example code gives an O(n2) worst-case running time.

```
void shell_sort(int A[], int size)

{

int i, j, increment, temp;

increment = size / 2;

while (increment > 0)
```

```
{
for (i=increment; i < size; i++)
{
j = i;
temp = A[i];
while ((j >= increment) && (A[j-increment] > temp))
{
A[j] = A[j - increment];
j = j - increment;
}
A[j] = temp;
}
if (increment == 2)
increment = 1;
else
increment = (int) (increment / 2.2);
}
}
```

Shell sort algorithm in Java

The Java implementation of Shell sort is as follows:

```
public static void shellSort(int[] a) {

for ( int increment = a.length / 2;

increment > 0;

increment = (increment == 2 ? 1 : (int) Math.round(increment / 2.2))) {

for (int i = increment; i < a.length; i++) {

for (int j = i; j >= increment && a[j - increment] > a[j]; j -= increment) {

int temp = a[j];

a[j] = a[j - increment];

a[j - increment] = temp;

}

}

}

}
```

Shell sort algorithm in Python

Here it is:

```
def shellsort(a):

def new_increment(a):

i = int(len(a) / 2)
```

```
yield i

while i != 1:

if i == 2:

i = 1

else:

i = int(numpy.round(i/2.2))

yield i

for increment in new_increment(a):

for i in xrange(increment, len(a)):

for j in xrange(i, increment-1, -increment):

if a[j - increment] < a[j]:

break

temp = a[j];

a[j] = a[j - increment]

a[j - increment] = temp

return a
```

6.2.2. Heap sort

(From Wikipedia, the free encyclopedia)

Heapsort is a comparison-basedsorting algorithm, and is part of the selection sort family. Although somewhat slower in practice on most

machines than a good implementation of quicksort, it has the advantage of a worst-case O(n log n) runtime. Heapsort is an in-place algorithm, but is not a stable sort.

Overview

Heapsort inserts the input list elements into a heap data structure. The largest value (in a max-heap) or the smallest value (in a min-heap) are extracted until none remain, the values having been extracted in sorted order. The heap's invariant is preserved after each extraction, so the only cost is that of extraction.

During extraction, the only space required is that needed to store the heap. In order to achieve constant space overhead, the heap is stored in the part of the input array that has not yet been sorted. (The structure of this heap is described at Binary heap: Heap implementation.)

Heapsort uses two heap operations: insertion and root deletion. Each extraction places an element in the last empty location of the array. The remaining prefix of the array stores the unsorted elements.

Variations

- The most important variation to the simple variant is an improvement by R.W.Floyd which gives in practice about 25% speed improvement by using only one comparison in each siftup run which then needs to be followed by a siftdown for the original child; moreover it is more elegant to formulate. Heapsort's natural way of indexing works on indices from 1 up to the number of items. Therefore the start address of the data should be shifted such that this logic can be implemented avoiding unnecessary +/- 1 offsets in the coded algorithm.
- Ternary heapsort uses a ternary heap instead of a binary heap; that is, each element in the heap has three children. It is more complicated to program, but does a constant number of times fewer swap and

comparison operations. This is because each step in the shift operation of a ternary heap requires three comparisons and one swap, whereas in a binary heap two comparisons and one swap are required. The ternary heap does two steps in less time than the binary heap requires for three steps, which multiplies the index by a factor of 9 instead of the factor 8 of three binary steps. Ternary heapsort is about 12% faster than the simple variant of binary heapsort.[citation needed]

- The smoothsort sorting algorithm is a variation of heapsort developed by Edsger Dijkstra in 1981. Like heapsort, smoothsort's upper bound is O(n log n). The advantage of smoothsort is that it comes closer to O(n) time if the input is already sorted to some degree, whereas heapsort averages O(n log n) regardless of the initial sorted state. Due to its complexity, smoothsort is rarely used.

Comparison with other sorts

Heapsort primarily competes with quicksort, another very efficient general purpose nearly-in-place comparison-based sort algorithm.

Quicksort is typically somewhat faster, due to better cache behavior and other factors, but the worst-case running time for quicksort is O(n2), which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation, creating a security risk. See quicksort for a detailed discussion of this problem, and possible solutions.

Thus, because of the O(n log n) upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort.

Heapsort also competes with merge sort, which has the same time bounds, but requires Ω(n) auxiliary space, whereas heapsort requires only a constant amount. Heapsort also typically runs more quickly in practice on machines with small or slow data caches. On the other hand, merge sort has several advantages over heapsort:

- Like quicksort, merge sort on arrays has considerably better data cache performance, often outperforming heapsort on a modern desktop PC, because it accesses the elements in order.
- Merge sort is a stable sort.
- Merge sort parallelizes better; the most trivial way of parallelizing merge sort achieves close to linear speedup, while there is no obvious way to parallelize heapsort at all.
- Merge sort can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Heapsort relies strongly on random access, and its poor locality of reference makes it very slow on media with long access times.

An interesting alternative to Heapsort is Introsort which combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

Pseudocode

The following is the "simple" way to implement the algorithm, in pseudocode, where swap is used to swap two elements of the array. Notice that the arrays are zero based in this example.

function heapSort(a, count) is

input: an unordered array a of length count

(first place a in max-heap order)

heapify(a, count)

end := count - 1

while end > 0 do

(swap the root(maximum value) of the heap with the last element of the heap)

swap(a[end], a[0])

(decrease the size of the heap by one so that the previous max value will

stay in its proper placement)

end := end - 1

(put the heap back in max-heap order)

siftDown(a, 0, end)

function heapify(a,count) is

(start is assigned the index in a of the last parent node)

start := count ÷ 2 - 1

while start ≥ 0 do

(sift down the node at index start to the proper place such that all nodes below

the start index are in heap order)

siftDown(a, start, count-1)

start := start - 1

(after sifting down the root all nodes/elements are in heap order)

function siftDown(a, start, end) is

input: end represents the limit of how far down the heap

to sift.

root := start

while root * 2 + 1 ≤ end do (While the root has at least one child)

child := root * 2 + 1 (root*2+1 points to the left child)

(If the child has a sibling and the child's value is less than its sibling's...)

if child < end and a[child] < a[child + 1] then

child := child + 1 (... then point to the right child instead)

if a[root] < a[child] then (out of max-heap order)

swap(a[root], a[child])

root := child (repeat to continue sifting down the child now)

else

return

The heapify function can be thought of as successively inserting into the heap and sifting up. The two versions only differ in the order of data processing. The above heapify function starts at the bottom and moves up while sifting down (bottom-up). The following heapify function starts at the top and moves down while sifting up (top-down).

function heapify(a,count) is

(end is assigned the index of the first (left) child of the root)

end := 1

while end < count

(sift up the node at index end to the proper place such that all nodes above

the end index are in heap order)

siftUp(a, 0, end)

end := end + 1

(after sifting up the last node all nodes are in heap order)

function siftUp(a, start, end) is

input: start represents the limit of how far up the heap to sift.

end is the node to sift up.

child := end

while child > start

parent := ⌊(child - 1) ÷ 2⌋

if a[parent] < a[child] then (out of max-heap order)

swap(a[parent], a[child])

child := parent (repeat to continue sifting up the parent now)

else

return

It can be shown that both variants of heapify run in O(n) time.[citation needed]

C-code

Below is an implementation of the "standard" heapsort (also called bottom-up-heapsort). It is faster on average (see Knuth. Sec. 5.2.3, Ex. 18) and even better in worst-case behavior (1.5n log n) than the simple heapsort (2n log n). The sift_in routine is first a sift_up of the free position followed by a sift_down of the new item. The needed data-comparison is only in the macro data_i_LESS_THAN_ for easy adaption.

This code is flawed - see talk page

}

```
/* Heapsort based on ideas of J.W.Williams/R.W.Floyd/S.Carlsson */

#define data_i_LESS_THAN_(other) (data[i] < other)

#define MOVE_i_TO_free { data[free]=data[i]; free=i; }

void sift_in(unsigned count, SORTTYPE *data, unsigned free_in,
SORTTYPE next)

{

unsigned i;

unsigned free = free_in;

// sift up the free node

for (i=2*free;i<count;i+=i)

{ if (data_i_LESS_THAN_(data[i+1])) i++;

MOVE_i_TO_free

}

// special case in sift up if the last inner node has only 1 child

if (i==count)

MOVE_i_TO_free

// sift down the new item next

while( ((i=free/2)>=free_in) && data_i_LESS_THAN_(next))

MOVE_i_TO_free

data[free] = next;

}
```

```
void heapsort(unsigned count, SORTTYPE *data)
{
unsigned j;
if (count <= 1) return;
data-=1; // map addresses to indices 1 til count
// build the heap structure
for(j=count / 2; j>=1; j--) {
SORTTYPE next = data[j];
sift_in(count, data, j, next);
}
// search next by next remaining extremal element
for(j= count - 1; j>=1; j--) {
SORTTYPE next = data[j + 1];
data[j + 1] = data[1]; // extract extremal element from the heap
sift_in(j, data, 1, next);
}
}
```

6.2.3. Quicksort

(From Wikipedia, the free encyclopedia)

Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare that, on average, makes

$\Theta(n \log n)$

(big O notation) comparisons to sort n items. However, in the worst case, it makes Θ(n2) comparisons. Typically, quicksort is significantly faster in practice than other

$\Theta(n \log n)$

algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices which minimize the possibility of requiring quadratic time.

Quicksort is a comparison sort and is not a stable sort.

The algorithm

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

1. Pick an element, called a pivot, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which are always sorted. The algorithm always terminates because it puts at least one element in its final place on each iteration (the loop invariant).

In simple pseudocode, the algorithm might be expressed as:

```
function quicksort(array)

var list less, pivotList, greater

if length(array) ≤ 1

return array

select a pivot value pivot from array

for each x in array

if x < pivot then add x to less

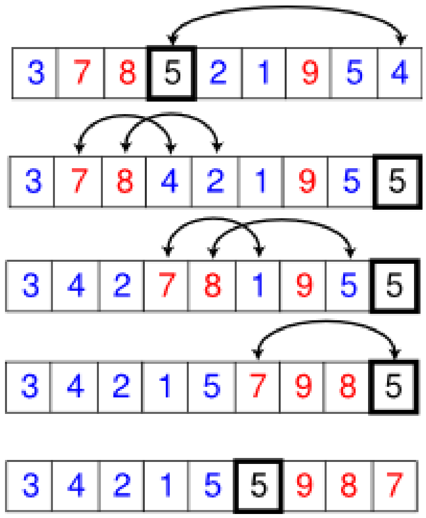
if x = pivot then add x to pivotList

if x > pivot then add x to greater

return concatenate(quicksort(less), pivotList, quicksort(greater))
```

Notice that we only examine elements by comparing them to other elements. This makes quicksort a comparison sort.

Version with in-place partition



Partition

In-place partition in action on a small list. The boxed element is the pivot element, blue elements are less or equal, and red elements are larger.

The disadvantage of the simple version above is that it requires Ω(n) extra storage space, which is as bad as mergesort (see big-O notation for the meaning of Ω). The additional memory allocations required can also drastically impact speed and cache performance in practical implementations. There is a more complicated version which uses an in-place partition algorithm and can achieve O(log n) space use on average for good pivot choices:

function partition(array, left, right, pivotIndex)

pivotValue := array[pivotIndex]

swap(array, pivotIndex, right) // Move pivot to end

storeIndex := left - 1

for i from left to right-1

if array[i] <= pivotValue

storeIndex := storeIndex + 1

swap(array, storeIndex, i)

swap(array, right, storeIndex+1) // Move pivot to its final place

return storeIndex+1

This form of the partition algorithm is not the original form; multiple variations can be found in various textbooks, such as versions not having the storeIndex. However, this form is probably the easiest to understand.

This is the in-place partition algorithm. It partitions the portion of the array between indexes left and right, inclusively, by moving all elements less than or equal to a[pivotIndex] to the beginning of the subarray, leaving all the greater elements following them. In the process it also finds the final position for the pivot element, which it returns. It temporarily moves the pivot element to the end of the subarray, so that it doesn't get in the way. Because it only uses exchanges, the final list has the same elements as the original list. Notice that an element may be exchanged multiple times before reaching its final place.

Once we have this, writing quicksort itself is easy:

function quicksort(array, left, right)

if right > left

select a pivot index (e.g. pivotIndex := left)

pivotNewIndex := partition(array, left, right, pivotIndex)

quicksort(array, left, pivotNewIndex-1)

quicksort(array, pivotNewIndex+1, right)

Parallelization

Like mergesort, quicksort can also be easily parallelized due to its divide-and-conquer nature. Individual in-place partition operations are difficult to parallelize, but once divided, different sections of the list can be sorted in parallel. If we have p processors, we can divide a list of n elements into p sublists in Θ(n) average time, then sort each of these in

$$\mathcal{O}\left(\frac{n}{p}\ \log\frac{n}{p}\right)$$

average time. Ignoring the O(n) preprocessing, this is linear speedup. Given Θ(n) processors, only O(n) time is required overall.

One advantage of parallel quicksort over other parallel sort algorithms is that no synchronization is required. A new thread is started as soon as a sublist is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.

Other more sophisticated parallel sorting algorithms can achieve even better time bounds. For example, in 1991 David Powers described a parallelized quicksort that can operate in O(log n) time given enough processors by performing partitioning implicitly[1].

Formal analysis

From the initial description it's not obvious that quicksort takes O(n log n)time on average. It's not hard to see that the partition operation, which simply loops over the elements of the array once, uses Θ(n) time. In versions that perform concatenation, this operation is also Θ(n).

In the best case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only (log n) nested calls before we reach a list of size 1. This means that the depth of the call tree is O(log n). But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls needs only O(n) time all together (each call has some constant overhead, but since there are only O(n) calls at

each level, this is subsumed in the O(n) factor). The result is that the algorithm uses only O(n log n) time.

An alternate approach is to set up a recurrence relation for T(n) factor), the time needed to sort a list of size n. Because a single quicksort call involves O(n) factor) work plus two recursive calls on lists of size n/2 in the best case, the relation would be:

$$\mathrm{T}(n) = \mathcal{O}(n) + 2\mathrm{T}(\frac{n}{2}).$$

The master theorem tells us that

$$\mathrm{T}(n) = \Theta(n \ \log \ n)$$

.

In fact, it's not necessary to divide the list this precisely; even if each pivot splits the elements with 99% on one side and 1% on the other (or any other fixed fraction), the call depth is still limited to (100log n), so the total running time is still O(n log n).

In the worst case, however, the two sublists have size 1 and n − 1, and the call tree becomes a linear chain of n nested calls. The ith call does

$$\mathcal{O}(n - i)$$

work, and

$$\sum_{i=0}^{n}(n - i) = \mathcal{O}(n^2)$$

. The recurrence relation is:

$$\mathrm{T}(n) = \mathcal{O}(n) \ + \ \mathrm{T}(1) \ + \ \mathrm{T}(n-1) = \mathcal{O}(n) \ + \ \mathrm{T}(n-1)$$

This is the same relation as for insertion sort and selection sort, and it solves to $T(n) = \Theta(n2)$.

Randomized quicksort expected complexity

Randomized quicksort has the desirable property that it requires only O(n log n)expected time, regardless of the input. But what makes random pivots a good choice?

Suppose we sort the list and then divide it into four parts. The two parts in the middle will contain the best pivots; each of them is larger than at least 25% of the elements and smaller than at least 25% of the elements. If we could consistently choose an element from these two middle parts, we would only have to split the list at most 2log2n times before reaching lists of size 1, yielding an O(n log n) algorithm.

Unfortunately, a random choice will only choose from these middle parts half the time. The surprising fact is that this is good enough. Imagine that you are flipping a coin over and over until you get k heads. Although this could take a long time, on average only 2k flips are required, and the chance that you won't get k heads after 100k flips is infinitesimally small. By the same argument, quicksort's recursion will terminate on average at a call depth of only 2log2n. But if its average call depth is O(log n), and each level of the call tree processes at most n elements, the total amount of work done on average is the product, O(n log n).

Average complexity

Even if we aren't able to choose pivots randomly, quicksort still requires only O(n log n) time over all possible permutations of its input. Because this average is simply the sum of the times over all permutations of the input divided by n factorial, it's equivalent to choosing a random permutation of the input. When we do this, the pivot choices are essentially random, leading to an algorithm with the same running time as randomized quicksort.

More precisely, the average number of comparisons over all permutations of the input sequence can be estimated accurately by solving the recurrence relation:

$$\mathrm{C}(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(\mathrm{C}(i) + \mathrm{C}(n-i-1)) = 2n\ln n = 1.39n\log_2 n.$$

Here, n − 1 is the number of comparisons the partition uses. Since the pivot is equally likely to fall anywhere in the sorted list order, the sum is averaging over all possible splits.

This means that, on average, quicksort performs only about 39% worse than the ideal number of comparisons, which is its best case. In this sense it is closer to the best case than the worst case. This fast average runtime is another reason for quicksort's practical dominance over other sorting algorithms.

C(n) = (n-1) + C(n/2) + C(n/2)

= (n-1) + 2C(n/2)

= (n-1) + 2((n/2) - 1 + 2C(n/4))

= n + n + 4C(n/4) - 1 - 2

= n + n + n + 8C(n/8) - 1 - 2 - 4

= ...

= kn + 2^kC(n/(2^k)) - (1 + 2 + 4 + ... + 2^(k-1))

where log2n > k > 0

= kn + 2^kC(n/(2^k)) - 2^k + 1

-> nlog2n + nC(1) - n + 1.

Space complexity

The space used by quicksort depends on the version used.

Quicksort has a space complexity of O(log n), even in the worst case, when it is carefully implemented such that

- in-place partitioning is used. This requires O(1).
- After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most O(log n) space. Then the other partition is sorted using tail-recursion or iteration.

The version of quicksort with in-place partitioning uses only constant additional space before making any recursive call. However, if it has made O(log n) nested recursive calls, it needs to store a constant amount of information from each of them. Since the best case makes at most O(log n) nested recursive calls, it uses O(log n) space. The worst case makes O(n) nested recursive calls, and so needs O(n) space.

We are eliding a small detail here, however. If we consider sorting arbitrarily large lists, we have to keep in mind that our variables like left and right can no longer be considered to occupy constant space; it takes O(log n) bits to index into a list of n items. Because we have variables like this in every stack frame, in reality quicksort requires O(log2n) bits of space in the best and average case and O(n log n) space in the worst case. This isn't too terrible, though, since if the list contains mostly distinct elements, the list itself will also occupy O(log n) bits of space.

The not-in-place version of quicksort uses O(n) space before it even makes any recursive calls. In the best case its space is still limited to O(n), because each level of the recursion uses half as much space as the last, and

$$\sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$

Its worst case is dismal, requiring

$$\sum_{i=0}^{n}(n-i+1)=\Theta(n^2)$$

space, far more than the list itself. If the list elements are not themselves constant size, the problem grows even larger; for example, if most of the list elements are distinct, each would require about O(log n) bits, leading to a best-case O(n log n) and worst-case O(n2 log n) space requirement.

Selection-based pivoting

A selection algorithm chooses the kth smallest of a list of numbers; this is an easier problem in general than sorting. One simple but effective selection algorithm works nearly in the same manner as quicksort, except that instead of making recursive calls on both sublists, it only makes a single tail-recursive call on the sublist which contains the desired element. This small change lowers the average complexity to linear or Θ(n) time, and makes it an in-place algorithm. A variation on this algorithm brings the worst-case time down to O(n) (see selection algorithm for more information).

Conversely, once we know a worst-case O(n) selection algorithm is available, we can use it to find the ideal pivot (the median) at every step of quicksort, producing a variant with worst-case O(n log n) running time. In practical implementations, however, this variant is considerably slower on average.

Competitive sorting algorithms

Quicksort is a space-optimized version of the binary tree sort. Instead of inserting items sequentially into an explicit tree, quicksort organizes them concurrently into a tree that is implied by the recursive calls. The algorithms make exactly the same comparisons, but in a different order.

The most direct competitor of quicksort is heapsort. Heapsort is typically somewhat slower than quicksort, but the worst-case running time is always

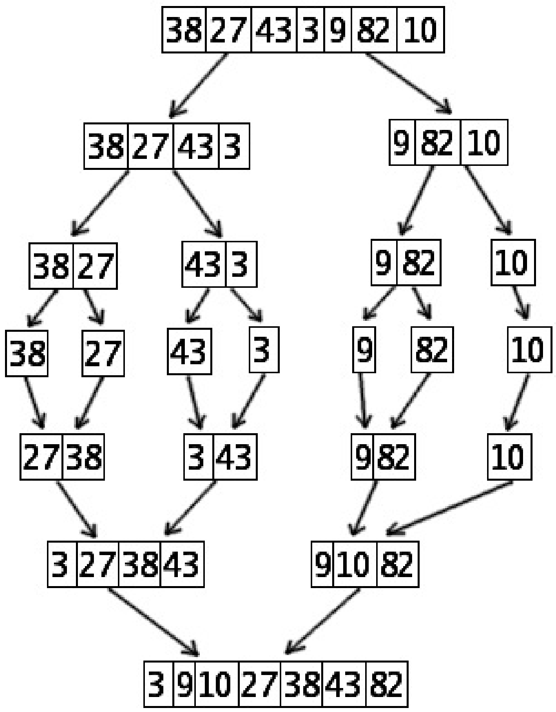
O(n log n). Quicksort is usually faster, though there remains the chance of worst case performance except in the introsort variant. If it's known in advance that heapsort is going to be necessary, using it directly will be faster than waiting for introsort to switch to it. Heapsort also has the important advantage of using only constant additional space (heapsort is in-place), whereas even the best variant of quicksort uses Θ(log n) space. However, heapsort requires efficient random access to be practical.

Quicksort also competes with mergesort, another recursive sort algorithm but with the benefit of worst-case O(n log n) running time. Mergesort is a stable sort, unlike quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Although quicksort can be written to operate on linked lists, it will often suffer from poor pivot choices without random access. The main disadvantage of mergesort is that, when operating on arrays, it requires Ω(n) auxiliary space in the best case, whereas the variant of quicksort with in-place partitioning and tail recursion uses only O(log n) space. (Note that when operating on linked lists, mergesort only requires a small, constant amount of auxiliary storage.)

6.2.4. Merge sort

(From Wikipedia, the free encyclopedia)

In computer science, merge sort or mergesort is an O(n log n) comparison-basedsorting algorithm. It is stable, meaning that it preserves the input order of equal elements in the sorted output. It is an example of the divide and conquer algorithmic paradigm. It was invented by John von Neumann in 1945.



Merge sort

A merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort.

Algorithm

Conceptually, merge sort works as follows:

1. Divide the unsorted list into two sublists of about half the size
2. Divide each of the two sublists recursively until we have list sizes of length 1, in which case the list itself is returned
3. Merge the two sublists back into one sorted list.

Mergesort incorporates two main ideas to improve its runtime:

1. A small list will take fewer steps to sort than a large list.

2. Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted (see the merge function below for an example implementation).

Example: Using mergesort to sort a list of integers contained in an array:

Suppose we have an array A with indices ranging from A’first to A’Last. We apply mergesort to A(A’first..A’centre) and A(centre+1..A’Last) - where centre is the integer part of (A’first + A’Last)/2. When the two halves are returned they will have been sorted. They can now be merged together to form a sorted array.

In a simple pseudocode form, the algorithm could look something like this:

function mergesort(m)

var list left, right, result

if length(m) ≤ 1

return m

else

var middle = length(m) / 2

for each x in m up to middle

add x to left

for each x in m after middle

add x to right

left = mergesort(left)

right = mergesort(right)

result = merge(left, right)

return result

There are several variants for the merge() function, the simplest variant could look like this:

function merge(left,right)

var list result

while length(left) > 0 and length(right) > 0

if first(left) ≤ first(right)

append first(left) to result

left = rest(left)

else

append first(right) to result

right = rest(right)

if length(left) > 0

append rest(left) to result

if length(right) > 0

append rest(right) to result

return result

C++ implementation

Here is an implementation using the STL algorithm std::inplace_merge to create an iterative bottom-up in-place merge sort:

```
#include <iostream>

#include <vector>

#include <algorithm>

#include <iterator>

int main()

{

std::vector<unsigned> data;

for(unsigned i = 0; i < 10; i++)

data.push_back(i);

std::random_shuffle(data.begin(), data.end());

std::cout << "Initial: ";

std::copy(data.begin(),data.end(),std::ostream_iterator<unsigned>
(std::cout," "));

std::cout << std::endl;

for(unsigned m = 1; m <= data.size(); m *= 2)

{

for(unsigned i = 0; i < data.size() - m; i += m * 2)

{

std::inplace_merge(
```

```
data.begin() + i,

data.begin() + i + m,

data.begin() + std::min<unsigned>(i + m * 2, (unsigned)data.size()));

}

}

std::cout << "Sorted: ";

std::copy(data.begin(),data.end(),std::ostream_iterator<unsigned>
(std::cout," "));

std::cout << std::endl;

return 0;

}
```

Analysis

In sorting n items, merge sort has an average and worst-case performance of O(n log n). If the running time of merge sort for a list of length n is T(n), then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem.

In the worst case, merge sort does approximately $(n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1)$ comparisons, which is between $(n \lg n - n + 1)$ and $(n \lg n + n + O(\lg n))$. [2]

For large n and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches α·n fewer than the worst case where

$$\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k + 1} \approx 0.2645$$

.

In the worst case, merge sort does about 39% fewer comparisons than quicksort does in the average case; merge sort always makes fewer comparisons than quicksort, except in extremely rare cases, when they tie, where merge sort's worst case is found simultaneously with quicksort's best case. In terms of moves, merge sort's worst case complexity is O(n log n)—the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.

Recursive implementations of merge sort make 2n - 1 method calls in the worst case, compared to quicksort's n, thus has roughly twice as much recursive overhead as quicksort. However, iterative, non-recursive, implementations of merge sort, avoiding method call overhead, are not difficult to code. Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in. Sorting in-place is possible but is very complicated, and will offer little performance gains in practice, even if the algorithm runs in O(n log n) time. In these cases, algorithms like heapsort usually offer comparable speed, and are far less complex.

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort as long as the merge operation is implemented properly.

As can be seen from the procedure MergeSort, there are some complaints. One complaint we might raise is its use of 2n locations; the additional n locations were needed because one couldn't reasonably merge two sorted sets in place. But despite the use of this space the algorithm must still work hard, copying the result placed into Result list back into m list on each call of merge . An alternative to this copying is to associate a new field of information with each key. (the elements in m are called keys). This field

will be used to link the keys and any associated information together in a sorted list (keys and related informations are called records). Then the merging of the sorted lists proceeds by changing the link values and no records need to moved at all. A field which contains only a link will generally be smaller than an entire record so less space will also be used.

Merge sorting tape drives

Merge sort is so inherently sequential that it's practical to run it using slow tape drives as input and output devices. It requires very little memory, and the memory required does not change with the number of data elements. If you have four tape drives, it works as follows:

1. Divide the data to be sorted in half and put half on each of two tapes
2. Merge individual pairs of records from the two tapes; write two-record chunks alternately to each of the two output tapes
3. Merge the two-record chunks from the two output tapes into four-record chunks; write these alternately to the original two input tapes
4. Merge the four-record chunks into eight-record chunks; write these alternately to the original two output tapes
5. Repeat until you have one chunk containing all the data, sorted --- that is, for log n passes, where n is the number of records.

For the same reason it is also very useful for sorting data on disk that is too large to fit entirely into primary memory. On tape drives that can run both backwards and forwards, you can run merge passes in both directions, avoiding rewind time.

Optimizing merge sort

This might seem to be of historical interest only, but on modern computers, locality of reference is of paramount importance in software optimization, because multi-level memory hierarchies are used. In some sense, main RAM can be seen as a fast tape drive, level 3 cache memory as a slightly

faster one, level 2 cache memory as faster still, and so on. In some circumstances, cache reloading might impose unacceptable overhead and a carefully crafted merge sort might result in a significant improvement in running time. This opportunity might change if fast memory becomes very cheap again, or if exotic architectures like the Tera MTA become commonplace.

Designing a merge sort to perform optimally often requires adjustment to available hardware, eg. number of tape drives, or size and speed of the relevant cache memory levels.

Typical implementation bugs

A typical mistake made in many merge sort implementations is the division of index-based lists in two sublists. Many implementations determine the middle index as outlined in the following implementation example:

```
function merge(int left, int right)

{

if (left < right) {

int middle = (left + right) / 2;

[...]
```

While this algorithm appears to work very well in most scenarios, it fails for very large lists. The addition of "left" and "right" would lead to an integer overflow, resulting in a completely wrong division of the list. This problem can be solved by increasing the data type size used for the addition, or by altering the algorithm:

```
int middle = left + ((right - left) / 2);
```

Note that the following two examples do not address the issue of integer overflow but dodge it under irrelevant efficiency claims

Probably faster, and arguably as clear is:

int middle = (left + right) >>> 1;

In C and C++ (where you don't have the >>> operator), you can do this:

middle = ((unsigned) (left + right)) >> 1;

See more information here:
http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

Comparison with other sort algorithms

Although heapsort has the same time bounds as merge sort, it requires only Θ(1) auxiliary space instead of merge sort's Θ(n), and is often faster in practical implementations. Quicksort, however, is considered by many to be the fastest general-purpose sort algorithm. On the plus side, merge sort is a stable sort, parallelizes better, and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only Θ(1) extra space, and the slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

As of Perl 5.8, merge sort is its default sorting algorithm (it was quicksort in previous versions of Perl). In Java, the Arrays.sort() methods use mergesort or a tuned quicksort depending on the datatypes and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.

Utility in online sorting

Mergesort's merge operation is useful in online sorting, where the list to be sorted is received a piece at a time, instead of all at the beginning (see online algorithm). In this application, we sort each new piece that is

received using any sorting algorithm, and then merge it into our sorted list so far using the merge operation. However, this approach can be expensive in time and space if the received pieces are small compared to the sorted list — a better approach in this case is to store the list in a self-balancing binary search tree and add elements to it as they are received.

Design Patterns for Sorting
In 1985, Susan Merritt proposed a new taxonomy for comparison-based sorting algorithms. At the heart of Merritt's thesis is the principle of divide and conquer. Merritt's thesis is potentially a very powerful method for studying and understanding sorting. However, the paper did not offer any concrete implementation of the proposed taxonomy. The following is our object-oriented formulation and implementation of Merritt's taxonomy.

The following discussion is based on the the SIGCSE 2001 paper by Nguyen and Wong, "Design Patterns for Sorting"[footnote].
D. Nguyen and S. Wong, "Design Patterns for Sorting," SIGCSE Bulletin 33:1, March 2001, 263-267

Merritt's Thesis
In 1985, Susan Merritt proposed that all comparison-based sorting could be viewed as "Divide and Conquer" algorithms.[footnote] That is, sorting could be thought of as a process wherein one first "divides" the unsorted pile of whatever needs to sorted into smaller piles and then "conquers" them by sorting those smaller piles. Finally, one has to take the the smaller, now sorted piles and recombines them into a single, now-sorted pile.
S. Merritt, "An Inverted Taxonomy of Sorting Algorithms," Comm. of the ACM, Jan. 1985, Volume 28, Number 1, pp. 96-99

We thus end up with a recursive definition of sorting:

- To sort a pile:
    - Split the pile into smaller piles
    - Sort the smaller piles
    - Join the sorted smaller piles into a single pile

We can see Merritt's recursive notion of sorting as a split-sort-join process in a pictoral manner by considering the general sorting process as a "black box" process that takes an unsorted set and returns a sorted set. Merritt's thesis thus contends that this sorting process can be described as a splitting followed by a sorting of the smaller pieces followed by a joining of the sorted pieces. The smaller sorting process can thus be similarly described. The base case of this recursive process is when the set has been reduced to a single element, upon which the sorting process cannot be broken down any more as it is a trivial no-op.
Animation of the Merritt Sorting Thesis (Click the "Reveal More" button)

Sorting can be seen as a recursive process that splits the unsorted items into multiple unsorted sets, sorts them and then rejoins the now sorted sets. When a set is reduced to a single element (blank boxes above), sorting is a trivial no-op.

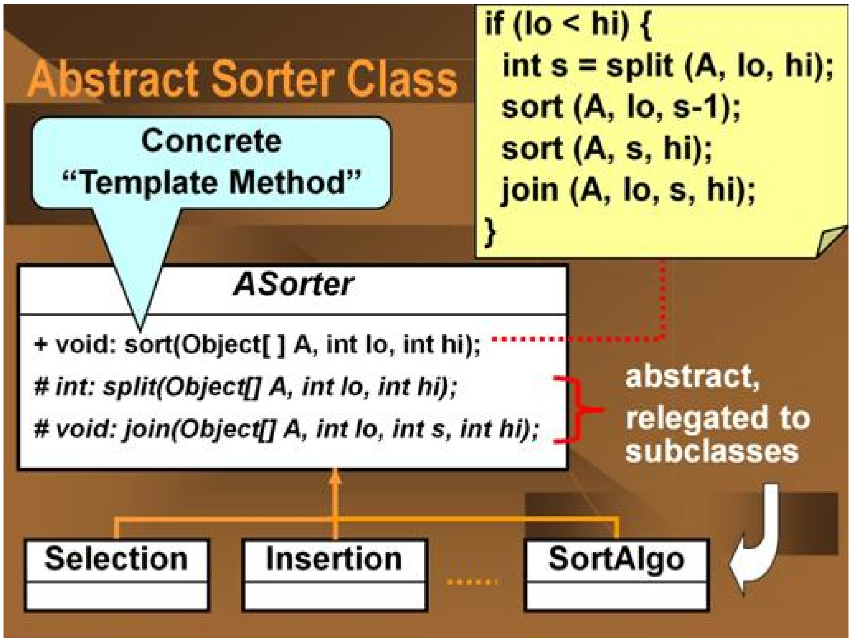
Merritt's thesis is potentially a very powerful method for studying and understanding sorting. In addition, Merritt's abstract characterization of sorting exhibits much object-oriented (OO) flavor and can be described in terms of OO concepts.

Capturing the Abstraction

So, how do we capture the abstraction of sorting as described by Merritt? Fundamentally, we have to recognize that the above description of sorting contains two distinct parts: the **invariant** process of splitting into sub-piles, sorting the sub-piles and joining the sub-piles, and the **variant** processes of the actual splitting and joining algorithms used.

Here, we will restrict ourselves to the process of sorting an array of objects, in-place -- that is, the original array is mutated from unsorted to sorted (as opposed to returning a new array of sorted values and leaving the original untouched). The `Comparator` object used to compare objects will be given to the sorter's constructor.

Abstract Sorter Class



The invariant sorting process is represented as an abstract class

Here, the invariant process is represented by the concrete `sort` method, which performs the split-sort-sort-join process as described by Merritt. The variant processes are represented by the abstract `split` and `join` methods, whose exact behaviors are indeterminate at this time.

Above the methods are defined as following:

`final void sort(Object [] A, int lo, int hi)` -- sorts the given unsorted array of objects, `A`, defined from index `lo` to index `hi`, inclusive. This method is implemented here and marked `final` to enforce its invariance with respect to the subclasses. It is this method that implements Merritt's split-sort-join process.

`abstract int split(Object [] A, int lo, int hi)` -- splits the given unsorted array of objects, `A`, defined from index `lo` to index `hi`, inclusive, into two adjacent sub-arrays. The returned index is the index of the first element of the upper sub-array. The implementation of this abstract method is in the sub-classes.

`abstract void join(Object [] A, int lo, int s, int hi)` -- joins two sorted adjacent sub-arrays of objects in the array `A`, where the lower sub-array is from index `lo` to index `s`, inclusive, and the upper sub-array is from index `s` to index `hi`, inclusive. The implementation of this abstract method is in the subclasses.

Here's the full code for the abstract `ASorter` class, the abstract superclass for all concrete sorters and the implementation of Merritt's template for sorting:

ASorter class

```
package sorter;

public abstract class ASorter
{
 protected AOrder aOrder;
 /**
  * The constructor for this class.
  * @param aOrder The abstract ordering strategy to be used by any subclass.
  */
 protected ASorter(AOrder aOrder)
 {
  this.aOrder = aOrder;
 }


 /**
  * Sorts by doing a split-sort-sort-join.  Splits the original array into
two subarrays,
  * recursively sorts the split subarrays, then re-joins the sorted subarrays
together.
  * This is the template method.  It calls the abstract methods split and
join to do
  * the work.  All comparison-based sorting algorithms are concrete
subclasses with
  * specific split and join methods.
  * @param A the array A[lo:hi] to be sorted.
  * @param lo the low index of A.
  * @param hi the high index of A.
  */
 public final void sort(Object[] A, int lo, int hi)
 {
      if (lo < hi)
      {
         int s = split (A, lo, hi);
         sort (A, lo, s-1);
         sort (A, s, hi);
         join (A, lo, s, hi);
      }
 }

 /**
  * Splits A[lo:hi] into A[lo:s-1] and A[s:hi] where s is the returned value
of this function.
  * @param A the array A[lo:hi] to be sorted.
  * @param lo the low index of A.
  * @param hi the high index of A.
  */
```

```
 protected abstract int split(Object[] A, int lo, int hi);

 /**
    * Joins sorted A[lo:s-1] and sorted A[s:hi] into A[lo:hi].
  * @param A A[lo:s-1] and A[s:hi] are sorted.
  * @param lo the low index of A.
  * @param hi the high index of A.
  */
 protected abstract void join(Object[] A, int lo, int s, int hi);

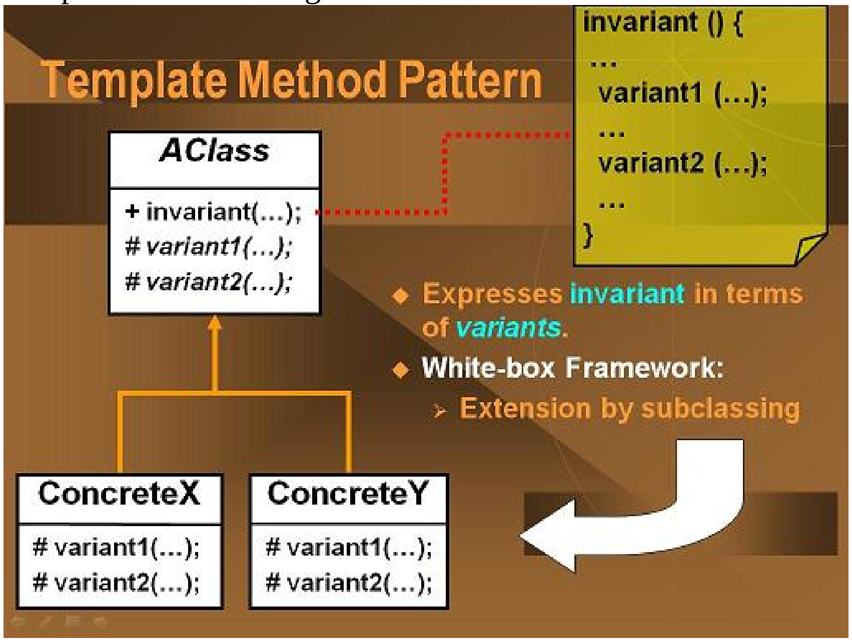
 /**
  * An accessor method for the abstract ordering strategy.
  * @param aOrder
  */
 public void setOrder(AOrder aOrder)
 {
  this.aOrder = aOrder;
 }
}
```

Note: `AOrder` is an abstract ordering operator whose concrete implementations define the binary ordering for the object being sorted. The examples below, only use the `AOrder.lt(Object x, Object y)` method, which returns `true` if `x < y`. The sorting framework could easily be modified to use `java.util.Comparator` instead with no loss of generality.

Template Design Pattern

The invariant sorting process as described by Merritt is an example of the Template Method Design Pattern.

Template Method Design Pattern



The Template Method Design Pattern describes an invariant concrete process in terms of variant, abstract

methods.

Here, the invariant process is represented by a concrete method of an abstract superclass. This concrete method's implementation is in terms of abstract methods of the same class. These abstract methods represent the variant processes and are implemented in the sub-classes. This type of class organization where the variant processes are relegated to sub-classes is also known as a **white box framework**.

Concrete Sorters

In order to create a sorter that can actually perform a sorting operation, we need to subclass the above `ASorter` class and implement the abstract `split` and `join` methods. It should be noted that in general, the `split` and `join` methods form a matched pair. One can argue that it is possible to write a universal join methods (a merge operation) but it would be highly inefficent in most cases.

Example:
Selection Sort
Tradionally, an in-place selection sort is performed by selecting the smallest (or largest) value in the array and placing it in the right-most location by either swapping it with the right-most element or by shifting all the in-between elements to the left. The selection and swapping/shifting process then repeated with the sub-array to the left of the newly placed element. This continues until only one element remains in the array. A selection sort is commonly used to do something like a sort group of people into ascending height.
Below is an animation of a traditional selection sort algorithm:
Traditional Selection Sort Algorithm

The extrema values are removed from an ever-shrinking unordered set and placed into the resulting sorted array. Here, the smallest values are removed from the left and placed to the right in the array.

In terms of the Merritt sorting paradigm, a selection sort can be broken down into a splitting process that is the same as the above selection process and a trivial join process. Looking at the above selection and swap/shift process, we see that it is describing a the splitting off of a single element, the smallest, from an array. The process repeats recursively until there is nothing more to split off. The sorting of a single element is a no-op, so after that the recursion rolls back out though the joining process. But the joining process is trivial, a no-op, because the elements are already in their corret positions. The beauty of Merritt's insight is the realize that by considering a no-op as an operational part of a process, all the different types of binary comparison-based sorting could be unified under a common framework. Below is an animation of a Merritt selection sort algorithm:

Merritt Selection Sort Process

/**

The splitting process splits off one element at a time, the smallest element, from the left and placed to the right in the array. The join process is a no-op because the elements are already in their correct places.

The code to implement a selection sorter is straightforward. One need only implement the `split` and `join` methods where the split method always returns the `lo+1` index because the smallest value in the (sub-)array has been moved to the index `lo` position. Because the bulk of the work is being done in the splitting method, selection sort is classified as an "hard split, easy join" sorting process. In the Java implementation of the SelectionSorter class below, the `split` method splits off the extrema (minimum, here) value from the sub-array, while the `join` method is a no-op.

SelectionSorter class

```
package sorter;

/**
```

```
 * A concrete sorter that uses the Selection Sort technique.
 */
public class SelectionSorter extends ASorter
{

 /**
  * The constructor for this class.
  * @param iCompareOp The comparison strategy to use in the sorting.
  */
 public SelectionSorter(AOrder iCompareOp)
 {
  super(iCompareOp);
 }
 /**
  * Splits A[lo:hi] into A[lo:s-1] and A[s:hi] where s is the returned value
of this function.
  * This method places the "smallest" value in the lo position and splits it
off.
  * @param A the array A[lo:hi] to be sorted.
  * @param lo the low index of A.
  * @param hi the high index of A.
  * @return lo+1 always
  */
 protected int split(Object[] A, int lo, int hi)
 {
     int s = lo;
      int i = lo + 1;
      // Invariant: A[s] <= A[lo:i-1].
      // Scan A to find minimum:
      while (i <= hi)
      {
         if (aOrder.lt(A[i], A[s]))
            s = i;
         i++; // Invariant is maintained.
      } // On loop exit: i = hi + 1; also invariant still holds;
        // this makes A[s] the minimum of A[lo:hi].

      // Swapping A[lo] with A[s]:
      Object temp = A[lo];
      A[lo] = A[s];
      A[s] = temp;
      return lo + 1;
  }

 /**
  * Joins sorted A[lo:s-1] and sorted A[s:hi] into A[lo:hi].
  * This method does nothing.  The sub-arrays are already in proper order.
  * @param A A[lo:s-1] and A[s:hi] are sorted.
  * @param lo the low index of A.
  * @param s
  * @param hi the high index of A.
  */
 protected void join(Object[] A, int lo, int s, int hi)
 {
 }
}
```

What's interesting to note here is what is missing from the above code. A tradional selection sort aalgorithm is implemented using a nested double loop, one to find the smallest value and one to repeatedly process the ever-shrinking unsorted sub-array. Notice that the above code only has a single loop, which coresponds to the inner loop of a traditional implementation. The outer loop is embodied in the recursive nature of the sort template method in the `ASorter` superclass.

Notice also that the selection sorter implementation does not include any explicit connection between the split and join operations nor does it contain the actual `sort` method. These are all contained in the concrete `sort` method of the superclass. We describe the `SelectionSorter` class as a **component** in a **framework** (technically a "white box" framework, as described above). Frameworks display **inverted control** where the components provide **services** to the framework. The framework itself runs the algorithms, here the high level, templated sorting process, and call upon the services provided by the components to fill in the necessary processing pieces, e.g. the split and join procedures.

Example:

Insertion Sort

Tradionally, an in-place insertion sort is performed by starting from one end of the arry, say the left end, and performing an in-order insertion of an element into the sub-array to its left. The next element to the right is then chosen and the insertion process repeated. At each insertion, the sorted sub-array on the left grows until encompasses the entire array. An insertion sort is a very typical way in which people will order a set of playing cards in their hand.

Below is an animation of a traditional insertion sort algorithm:

Traditional Insertion Sort Algorithm

Starting from the left, elements from the immediate right are inserted into a growing sub-array to the left.

In the Merrit paradigm, the insertion sort first splits the array or sub-array into two pieces simply by separating the right-most element. Recursively, the splitting process proceeds to from the right to the left until a single element is left in the sub-array. Sorting a one element array is a no-op, so then the recursion unwinds with the join process. The join process combines each single split-off element with its sorted sub-array partner to its left by performing an in-order insertion. This proceeds as the recusion unwinds until the entire array is fully sorted. In contrast to the selection sort, the bulk of the work is being done in the join method, hence classifying insertion sort as an "easy split, hard join" sorting process.

Below is an animation of a Merritt insertion sort algorithm:

Merritt Insertion Sort Process

The right-most elements are first split-off one by one, starting at the right and moving left. The split-off elements are then joined by performing an in-order insertion to the left, starting at the left.

In the Java implementation of the selection sorter below, the `split` method simply splits off the right-most element of the sub-array. The `join` method performs an in-order insertion of the single split-off element into the larger sub-array to its left.

InsertionSorter class

```
package sorter;

/**
 * A concrete sorter that uses the Insertion Sort technique.
 */
public class InsertionSorter extends ASorter
{
```

```
 /**
  * The constructor for this class.
  * @param iCompareOp The comparison strategy to use in the sorting.
  */
 public InsertionSorter(AOrder iCompareOp)
 {
  super(iCompareOp);
 }
 /**
  * Splits A[lo:hi] into A[lo:s-1] and A[s:hi] where s is the returned value
of this function.
  * This simply splits off the element at index hi.
  * @param A the array A[lo:hi] to be sorted.
  * @param lo the low index of A.
  * @param hi the high index of A.
  * @return hi always.
  */
 protected int split(Object[] A, int lo, int hi)
 {
  return (hi);
 }

 /**
  * Joins sorted A[lo:s-1] and sorted A[s:hi] into A[lo:hi].   (s = hi)
  * The method performs an in-order insertion of A[hi] into the A[lo, hi-1]
  * @param A A[lo:s-1] and A[s:hi] are sorted.
  * @param lo the low index of A.
  * @param s
  * @param hi the high index of A.
  */
 protected void join(Object[] A, int lo, int s, int hi)
 {
      int j = hi; // remember s == hi.
      Object key = A[hi];
      // Invariant: A[lo:j-1] and A[j+1:hi] are sorted and key < all
elements of A[j+1:hi].
      // Shifts elements of A[lo:j-1] that are greater than key to the
"right" to make room
      // for key.
      while (lo < j && aOrder.lt(key, A[j-1]))
      {
         A[j] = A[j-1];
         A[j-1] = key;
         j = j - 1;     // invariant is maintained.
      }   // On loop exit: j = lo or A[j-1] <= key. Also invariant is still
true.
 //     A[j] = key;
 }
}
```

Exercise:

Problem:

The authors were once challenged that the Merritt template-based sorting paradigm could not be used to describe the Shaker Sort process (a bidirectional Bubble or Selection sort). See for instance, http://en.wikipedia.org/wiki/Cocktail_sort. However, it can be done is a very straightforward manner. There are a number of viable solutions. Hint: think about the State Design Pattern.

Solution:

The solution is left to the student but is available from the authors if proof of non-student status is provided.

For more examples, please see download the demo code. Please note that the ShakerSort code is disabled due to its use as a student exercise.

Sorting an Array
An introduction and example of sorting an array within the C++ programming language.

Overview

Sorting is the process through which data are arranged according to their values. There are several sorting algorithms or methods that can be used to sort data. Some include:

1. Bubble
2. Selection
3. Insertion

We will not be covering the selection or insertion sort methods in this module.

"The bubble sort is an easy way to arrange data in ascending or descending order. If an array is sorted in ascending order, it means the values in the array are stored from lowest to highest. If values are sorted in descending order, they are stored from highest to lowest. Bubble sort works by comparing each element with its neighbor and swapping them it they are not in the desired order."[footnote]
Tony Gaddis, Judy Walters and Godfrey Muganda, Starting Out with C++ Early Objects Sixth Edition (United States of America: Pearson – Addison Wesley, 2008) 569.

There are several different methods of bubble sorting and some methods are more efficient than others. Most use a pair of nested loops or iteration control structures. One method sets a flag that indicates that the array is sorted, then does a pass and if any elements are exchanged (switched); it sets the flag to indicate that the array is not sorted. It is executed until it makes a pass and nothing is exchanged.

This bubble sort sets a flag that indicates that the array is sorted (that is it does not need more sorting), then does a pass and if any elements are exchanged (switched); it sets the flag to indicate that the array is not sorted (that is it needs more sorting). The outer do while loop is executed until the inner for loop makes a pass and nothing is exchanged.

Here is some color highlighted C++ code from **Demo_Sort_Array_Function.cpp**

```
do
   {
   moresortneeded = false;
   for(int i = 0; i < array_size - 1; i++)
     {
     if(things[i] > things[i+1])
       {
       temp = things[i];
       things[i] = things[i+1];
       things[i+1] = temp;
       moresortneeded = true;
       }
     }
   }
 while(moresortneeded);
```

The bubble sort gets its name from the lighter bubbles that move or "bubble up" to the top of a glass of soda pop. We move the smaller elements of the array to the top as the larger elements move to the bottom of the array. This can be viewed from a different perspective. Using an Italian salad dressing with oil, water and herbs; once shaken you can either:

1. envision the lighter oil rising to the top; **OR**
2. envision the heaver water and herbs sinking to the bottom

Either way is correct and this version of the code simply demonstrates the sinking to the bottom the heaver or larger elements of the array.

Bubble sorting is demonstrated in the demo file provided, thus you need to study this material in conjunction with the demo program.

Demonstration Program in C++

Creating a Folder or Sub-Folder for Source Code Files

Depending on your compiler/IDE, you should decide where to download and store source code files for processing. Prudence dictates that you create these folders as needed prior to downloading source code files. A suggested sub-folder for the **Bloodshed Dev-C++ 5 compiler/IDE** might be named:

- Demo_Programs

If you have not done so, please create the folder(s) and/or sub-folder(s) as appropriate.

Download the Demo Program

Download and store the following file(s) to your storage device in the appropriate folder(s). Following the methods of your compiler/IDE, compile and run the program(s). Study the source code file(s) in conjunction with other learning materials. You may need to right click on the link and select "Save Target As" in order to download the file.

Download from Connexions: Demo_Sort_Array_Function.cpp

Download from Connexions: Demo_Farm_Acres_Input.txt

Definitions

sorting
: Arranging data according to their values.

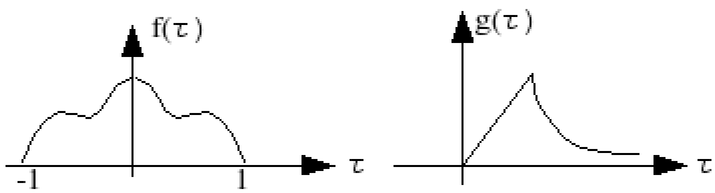
bubble sort
: A method of swapping array members until they are in the desired sequence.

Graphical Convolution Algorithm

$$c(t) = \int_{-\infty}^{\infty} f(\tau) g(t-\tau)\ \mathrm{d}\ \tau$$

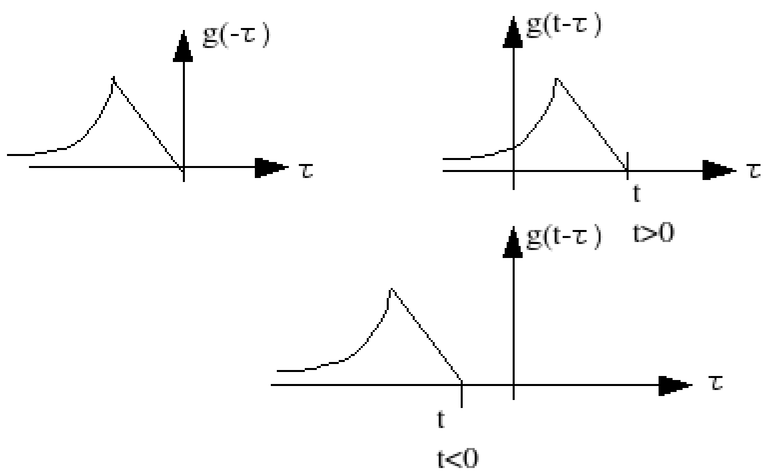
Step One

Plot $f(\tau)$ and $g(\tau)$ as functions of τ



Step Two

Plot $g(t-\tau)$ by reflecting $g(\tau)$ over the 'y-axis' (run time backwards) and then shifting right by t.

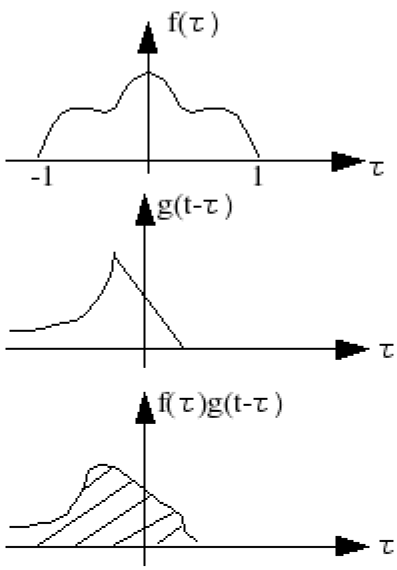


Step Three

For one value of 't' mutiply $f(\tau)g(t-\tau)$ and compute area underneath the curve to get $c(t)$. Area underneath

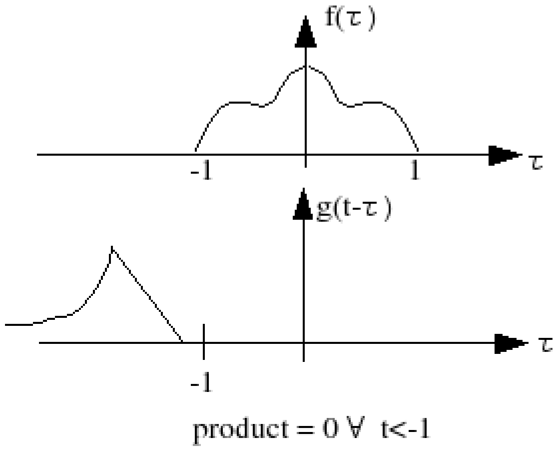
Equation:

$$= \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,\mathrm{d}\,\tau = c(t)$$



Step Four

Repeat for all 't' to get $c(t)$ for all t. Usually we will just have to consider several ranges of t.



Step Five

Reality check: Does your answer actually make sense?

Remark

Since,
Equation:

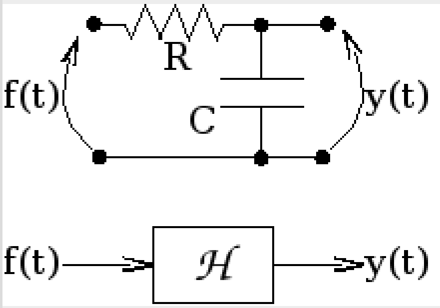
$$\begin{aligned} c(t) &= \int_{-\infty}^{\infty} f(\tau) g(t-\tau)\, \mathrm{d}\,\tau \\ &= \int_{-\infty}^{\infty} g(\tau) f(t-\tau)\, \mathrm{d}\,\tau \end{aligned}$$

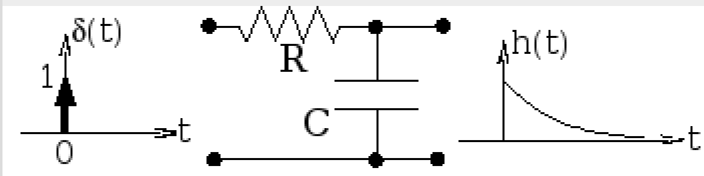
you can flip and shift either f or g. It is easier to flip and shift the 'simpler' of the two.

[missing_resource:]

Note: Everyone is overwhelmed by convolution at first! Just practise and it will become second nature. Do examples 2.6 to 2.8 in Lathi!

Example:
Recall





Now compute output $y(t)$ for a step input $f(t)u(t)$

Solution

System is LTI with impulse response $h(t)$, so use convolution integral

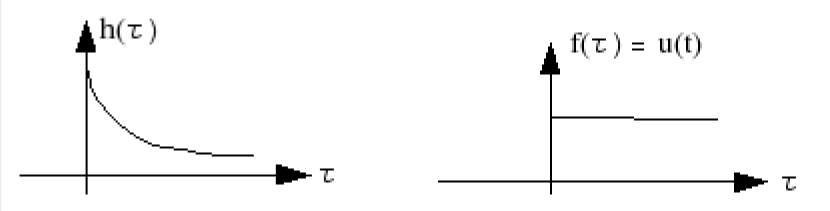
$$y(t) = \int_{-\infty}^{\infty} f(\tau)h(t-\tau)\ \mathrm{d}\ \tau$$

Since, $f(\tau)$ is simpler, we rewrite it as

$$\int_{-\infty}^{\infty} h(\tau)f(t-\tau)\ \mathrm{d}\ \tau$$

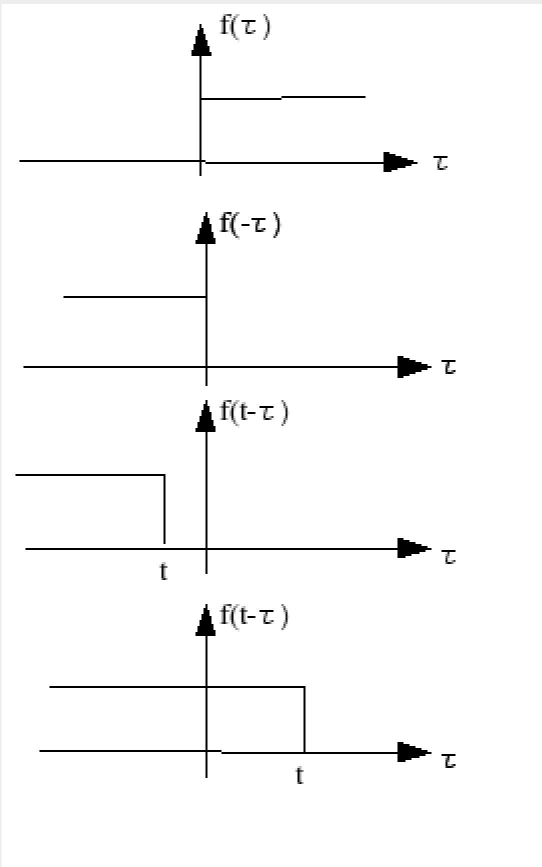
Step 1

Plot things



Step 2

Do the flip and shift.

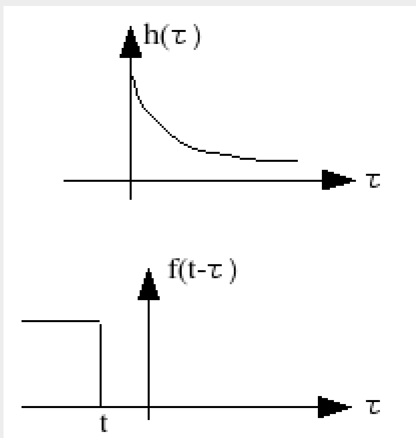


Step 3 & 4

Multiply and integrate.

Case 1

For, $t < 0$

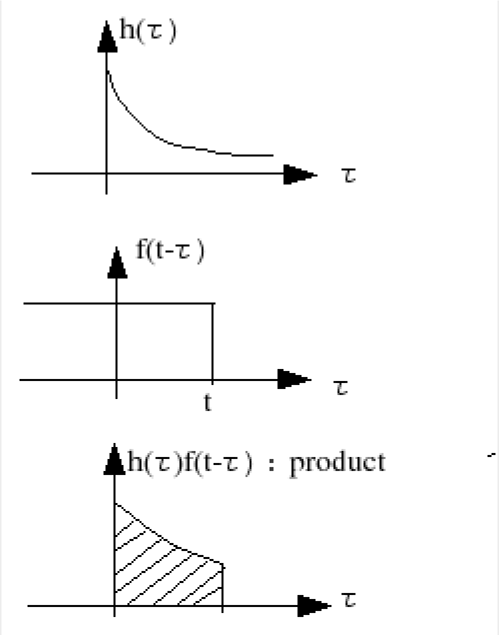


From the fact stated in the caption,

$$\int_{-\infty}^{\infty} h(\tau) f(t-\tau) \, \mathrm{d}\,\tau = y(t) = 0 \forall t : (t < 0)$$

Case 2

For $t \geq 0$

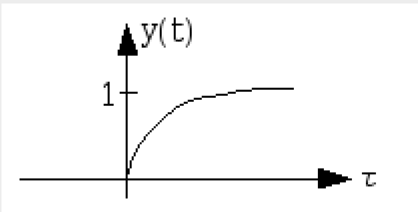


Equation:

$$\begin{aligned} y(t) &= \int_{-\infty}^{\infty} h(\tau) f(t-\tau) \, \mathrm{d}\, \tau \\ &= \int_0^t \frac{1}{RC} e^{-\frac{\tau}{RC}} \, \mathrm{d}\, \tau \\ &= \frac{RC}{RC} e^{-\frac{\tau}{RC}} \Big|_{\tau=0}^{t} \\ &= 1 - e^{-\frac{t}{RC}} \end{aligned}$$

Answer

$$y(t) = \begin{cases} 0 \text{ if } t < 0 \\ 1 - e^{\frac{-t}{RC}} \text{ if } t \geq 0 \end{cases}$$



Step 5

Do a reality check: As t tends to ∞ what happens? As t tends to −∞ what happens?

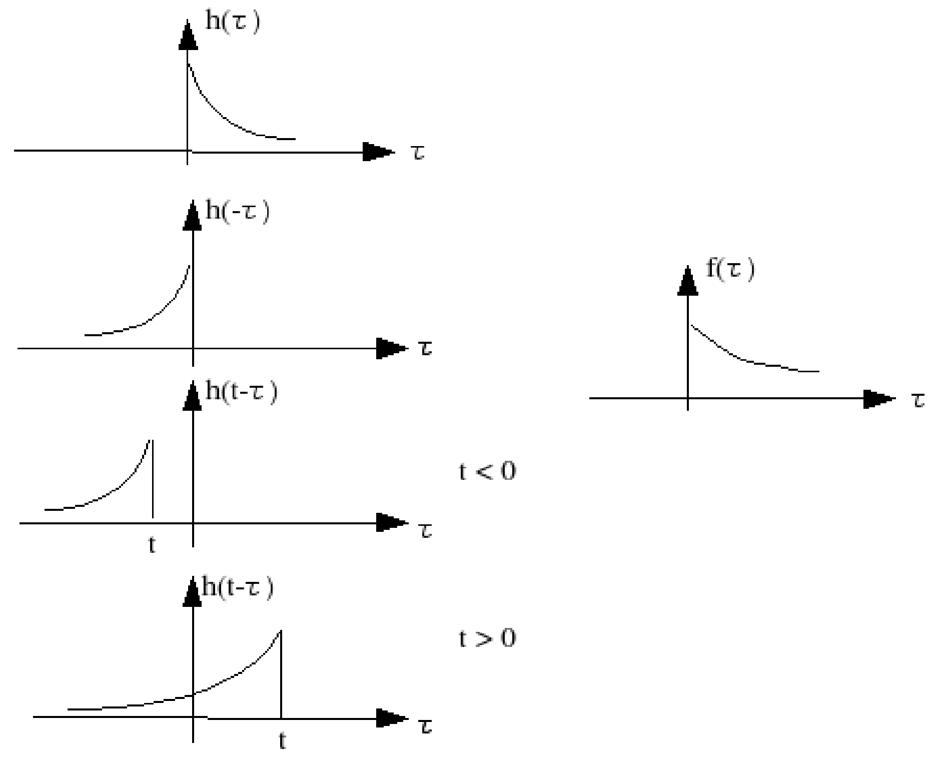
Example:
The input is $f(t) = e^{-t}$ and the impulse response is $h(t) = e^{-(2t)}$. Compute the $y(t)$.

Solution

We are given input and impulse response. So ride the convolution convoy!

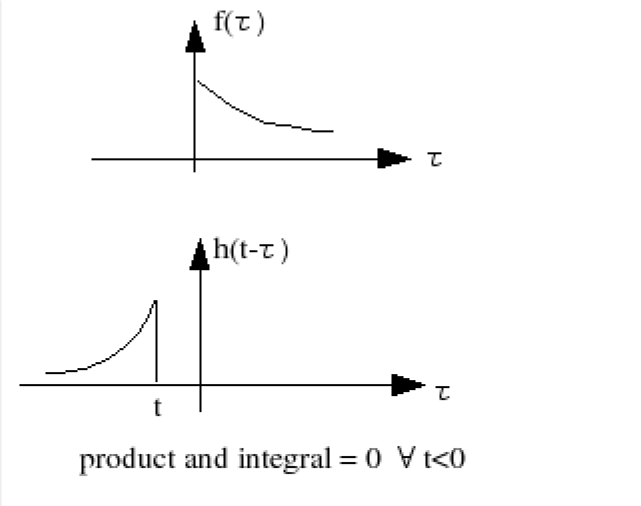
$$y(t) = \int_{-\infty}^{\infty} f(\tau)h(t-\tau)\,\mathrm{d}\,\tau$$

Both the functions are equally simple, so we flip and shift $h(t)$



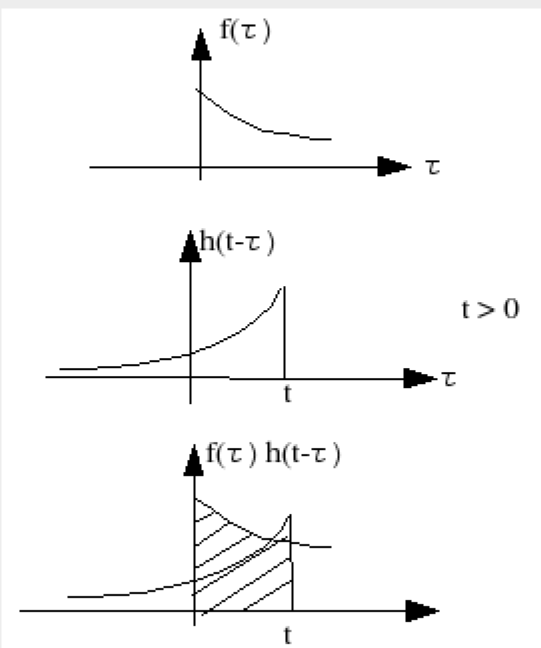
Case 1

Again $y(t) = 0$ for all $t < 0$



Case 2

For $t \geq 0$



Equation:

$$
\begin{aligned}
y(t) &= \int_{-\infty}^{\infty} f(\tau)h(t-\tau)\ \mathrm{d}\ \tau \\
&= \int_0^t f(\tau)h(t-\tau)\ \mathrm{d}\ \tau \\
&= \int_0^t e^{-\tau} e^{(-2)(t-\tau)}\ \mathrm{d}\ \tau \\
&= \int_0^t e^{-(2t)} e^{\tau}\ \mathrm{d}\ \tau \\
&= e^{-(2t)} \int_0^t e^{\tau}\ \mathrm{d}\ \tau \\
&= e^{-(2t)} e^{\tau}\big|_{\tau=0}^{t} \\
&= e^{-(2t)} \left(e^{t}-1\right)
\end{aligned}
$$

$$
y(t) = e^{-t} - e^{-(2t)} \forall t : (t \geq 0)
$$

Combine Case 1 and 2

Equation:

$$
\begin{aligned}
y(t) &= \begin{cases} 0 \text{ if } t < 0 \\ e^{-t} - e^{-(2t)} \text{ if } t \geq 0 \end{cases} \\
&= \left(e^{-t} - e^{-(2t)}\right) u(t)
\end{aligned}
$$

Algorithm Overview

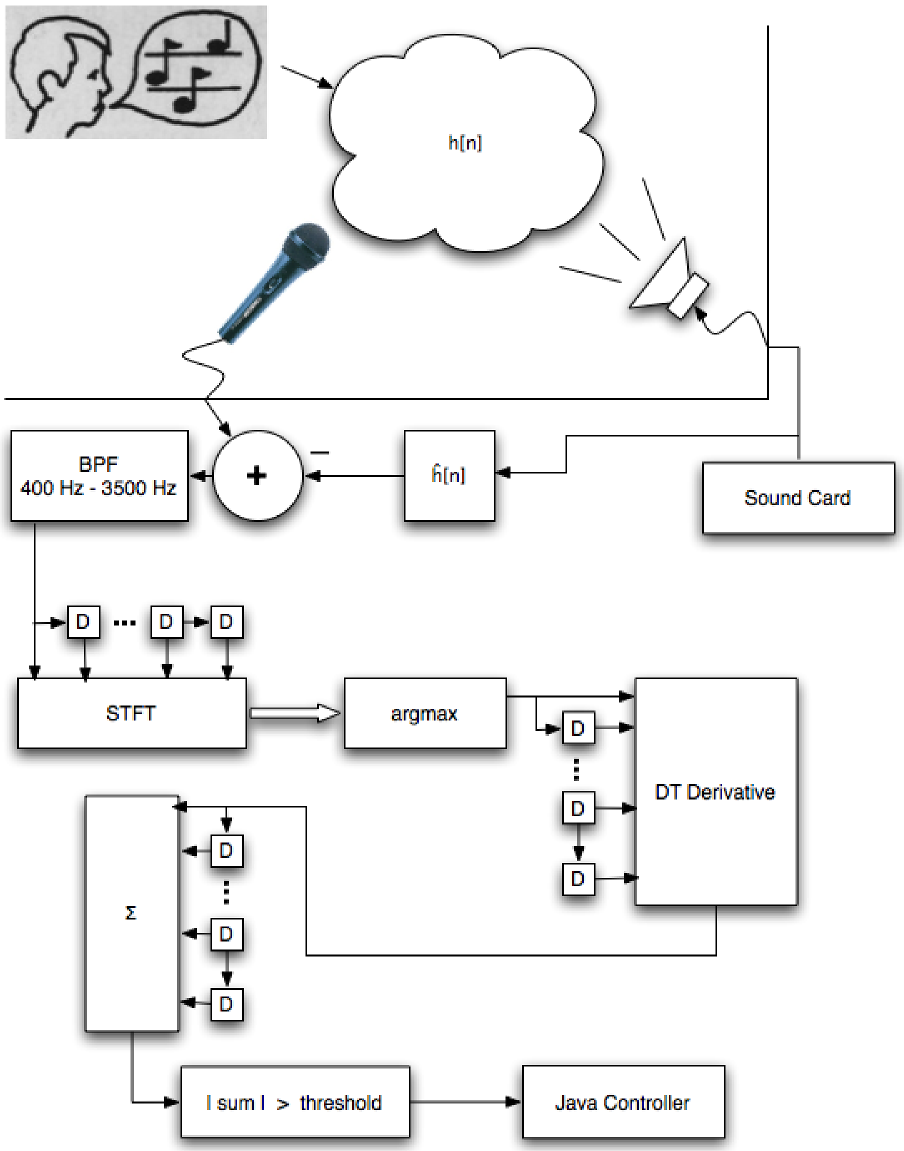
Algorithm Overview

The first step in detecting a signal is to input it into the system. Since we are using an audio signal, a microphone is the obvious choice. However, the desired control signal is not the only sound in the room. There is also the music that is being played through the speakers as well as outside noise. Unfortunately, there is not much that can be done about the random noise that is present in the room. However, there are tools available that allow us to minimize the interference caused by the music. Specifically, we can use an adaptive filter to mimic the room's effect on the output of the sound card, providing us with an estimate of the music's contribution to the signal received by the microphone. We can subtract the microphone's signal from this estimate in order to obtain—hopefully—only the whistle.

Figure 1 shows the block diagram of our system. All of these components fall into four main categories:

1. Signal acquisition
2. Whistle isolation
3. Whistle frequency analysis
4. iTunes interface

The acquisition phase is the top portion of the diagram, the whistle isolation system is represented by the $\hat{h}$ box and the band pass filter, the rest of the diagram (except for the Java controller) comprise the whistle analysis phase, and the Java controller is the iTunes interface.
Our System's Block Diagram



In our system, the sound is output through the speaker, and the microphone receives the music and whistles while the sound card receives the audio without the room affecting it. We then remove the music and process the whistle.

After isolating the whistle, we apply a band pass filter whose pass band corresponds to common whistle frequencies in order to remove extraneous noise outside of these whistle frequencies.

We then take the Short Time Fourier Transform in order see how the frequency components of the whistle change over time. If the frequency of the whistle is increasing iTunes should advance to the next track, and a decreasing frequency will skip to the previous track. To accomplish this, we examine the frequency with maximum power (the argmax in the figure below) and accumulate several readings of this frequency. In order to see if this function is increasing or decreasing we take the derivative and examine its average value. If the average value is positive, the function must have been increasing and the whistle must have been from high to low frequencies.